Client-Server Visualization of City Models through Non Photorealistic Rendering

Jean-Charles Quillet, Gwenola Thomas, Jean-Eudes Marvie

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

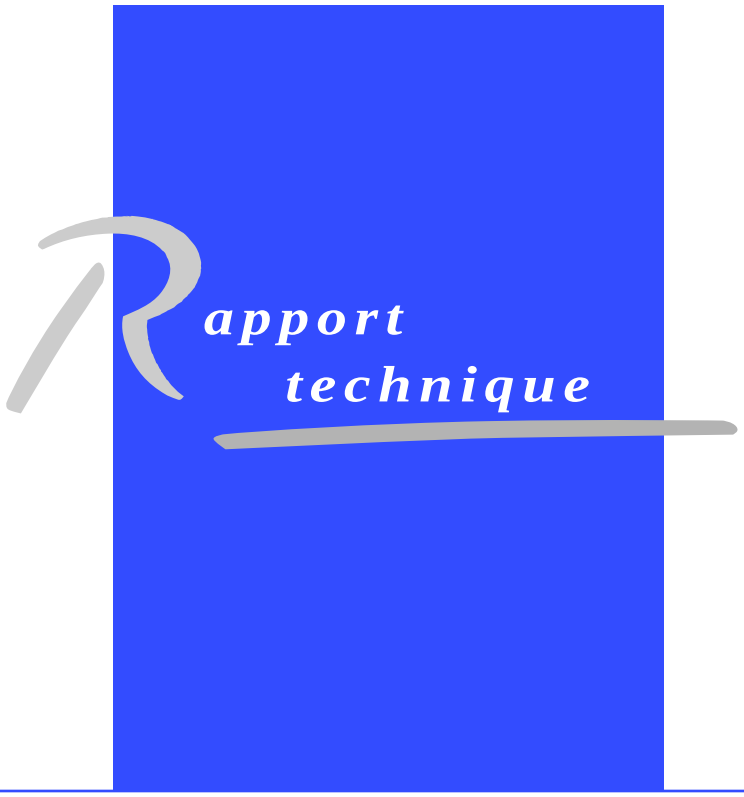
Client-Server Visualization of City Models through Non Photorealistic Rendering

Jean-Charles Quillet — Gwenola Thomas — Jean-Eudes Marvie

N° 0313

September 23, 2005

Thème COG



ISSN 0249-0803

Client-Server Visualization of City Models through Non Photorealistic Rendering

Jean-Charles Quillet , Gwenola Thomas , Jean-Eudes Marvie

Thème COG — Systèmes cognitifs
Projet IPARLA

Rapport technique n° 0313 — September 23, 2005 — 22 pages

Abstract:

In this report, we present a method allowing to visualize and interact with massive virtual urban environments adapted to mobile and connected devices like PDA or mobile phones. These devices have a very limited computational power and a restricted display area. We address these constraints using two optimizations. The first one is locale, we propose a simplified representation of building' facades as an alternative for textured rendering. Feature lines are extracted from facade images and vectorised. The resulting representation is more compact than the original image and allows to optimize transmission and rendering costs. As a global optimization, we split up the urban scenery in cells using the Magellan framework. Then we compute visibility relationships between the cells in order to stream the model.

Key-words: Non-photorealistic rendering, line based rendering, mobile and connected devices, client-server, streaming

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 —Télécopie : +33 1 72 92 59 ??

Visualisation client serveur d'environnements urbains non photoréalistes

Résumé :

Dans ce rapport, nous présentons une méthode permettant la visualisation et l'interaction dans de gros environnements urbains virtuels adaptée aux plateformes mobiles telles que les PDAs ou les téléphones portables. Ces terminaux disposent de capacités de calcul limitées et d'une surface d'affichage réduite. Nous adressons ces contraintes par deux optimisations. La première optimisation est locale; nous proposons une représentation simplifiée des façades de bâtiments comme alternative aux textures. Les traits caractéristiques d'images de façade sont extraits et vectorisés. La représentation obtenue est plus compacte que l'image originale et permet d'optimiser les coûts de transmission et de rendu. La seconde optimisation est globale; l'environnement urbain virtuel est découpé en cellules à l'aide de la plateforme Magellan. Puis nous calculons les relations de visibilité entre les cellules en vue d'une diffusion du modèle en flux tendu.

Mots-clés : Rendu non-photoréaliste, rendu basé ligne, terminaux mobiles communiquants, client-serveur, streaming

Contents

1 Introduction

Our objective is to propose some modeling and rendering tools for the interactive visualization of complex virtual urban environments on small devices. Wishing to be able to handle massive databases, we place ourself in client-server configuration. The server disposes of the whole geometry and sends on demand to the client a subset of the model. Visualisation and interaction are then initiated by the client. From the small display size and limited computation performances result strong constraints we want to solve using non-photorealistic rendering (NPR) techniques.

Usually, the buildings of a virtual city are simple textured blocks which are constructed from extrusions of their footprints. The textures are photographs of real building facades. The visual quality of resulting cities is satisfactory if some photographs of good resolution are used. In our case, transmission of high resolution textures can be prohibitive due to network bandwidth and high rendering cost on the client side.

As an answer to this problem, we propose to use, as an alternative for textured rendering, a set of feature lines which defines a given facade. These lines are stored in a vectorial form and thus allow to decrease significantly the size of a building representation. We have settled a whole operational pipeline able to extract the feature lines from photographs or texture images. And a second for streaming and rendering such models on mobile devices.

2 Overview

The figure 1 presents a general overview of the process. The first part consists in the modeling and optimisation of the 3D model. We first extrude the city footprints to get the first model. Feature lines extracted from the textures are then used to describe the facade into the geometric model. As a result we obtain a NPR database. The construction of this database is described in section 3.

The next part consists in optimizing the database in order to stream it. For this point we relied mainly on the Magellan [7, Part1] framework. The main purpose of Magellan is to allow easy and fast development of new solutions for the remote visualization of 3D models.

The NPR database needs first to be split up in connected cells, so the server is able to send to the client only the cells it requires. For further optimisation, visibility relationships between the cells are computed. As a result only the visible geometry is sent to the client and so rendered. The related processes are described in section 5.

Finally, the algorithms used by the client-server application are explained in section 5 and the results of our approach are discussed in section 6.

3 NPR city modeling

3.1 Facade transformation

The whole image-processing pipeline is shown on figure 2. Pictures of buildings are used at the start of the system. Feature lines are extracted using edge detector filters. Extracted contours can be seen as pixel chains which are then vectorized into poly-lines. This procedure allows to reduce the size of the data as well as to make the representation independent from the resolution. Then, this set of lines is used to build a 3D scene composed by a set of blocks one for each building. The lines are viewed in the 3D scene in the local block coordinate system .

For the first two stages (edge detection and polygonal approximation), we explored existing techniques and chose the methods adapted to our problem. For easy use, proposed filters have been implemented as plugins for The Gimp[1], a widely used image processing tool.

3.1.1 Automatic extraction of feature lines

We seek to obtain the feature lines from pictures of buildings. Those feature lines are represented by contours and discontinuities within the image. The first stage thus consists in extracting these contours. Once obtained, they are chained and then vectorized. Finally some heuristics proper to the images we work on are applied in order to simplify the obtained model.

Working on vectorial data offers two main advantages:

[1] GNU Image Manipulation Program `http://www.gimp.org/`

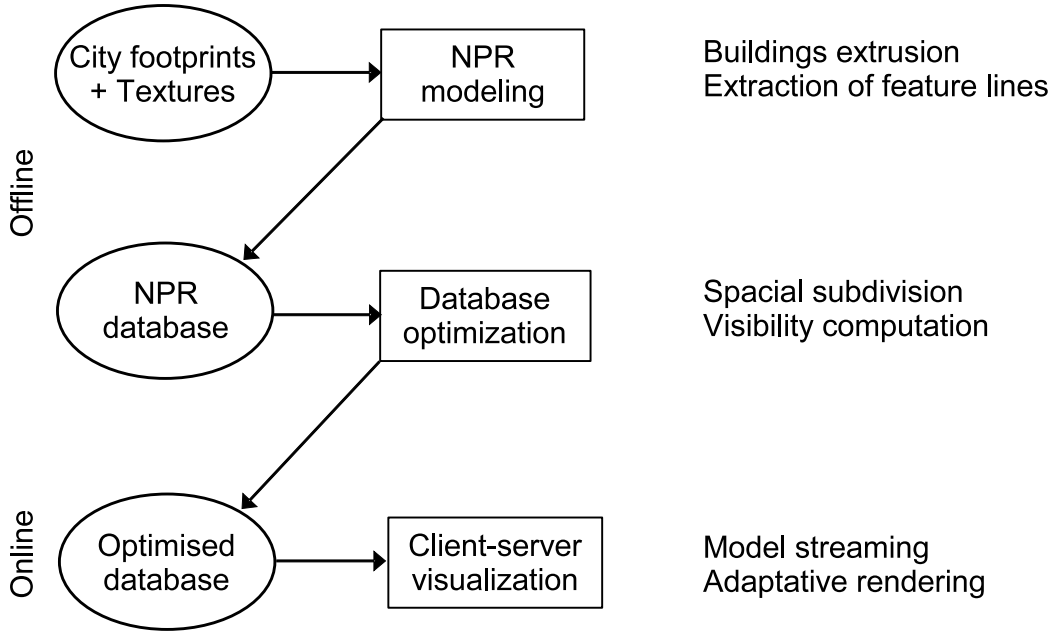


Figure 1: General overview. Offline processes consist in modeling and preparing the database to be used in a client-server application (online). The feature lines are extracted from the textures. The geometry is split up and the visibility relationship computed.

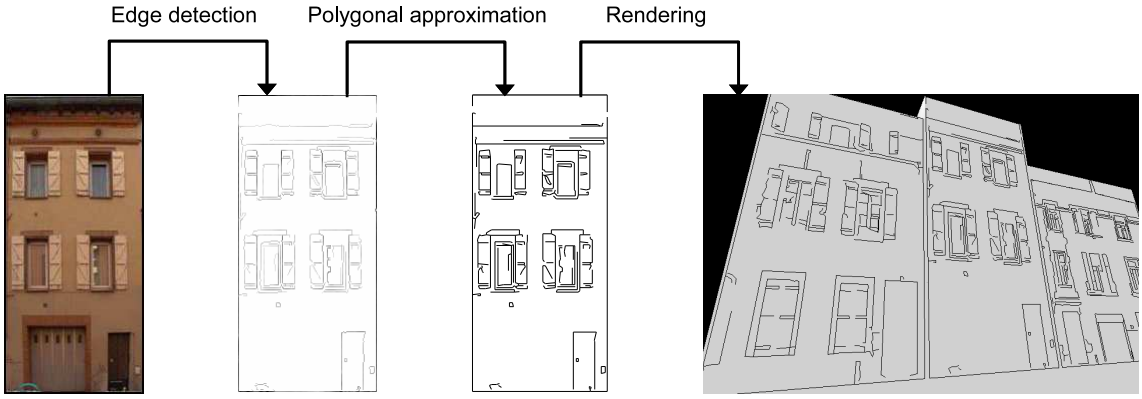


Figure 2: Production pipeline of stroke-based buildings. The edges of facade images are extracted and vectorized. These vector data are used instead of textures to represent the facade in the final 3D modele.

- The size of the data is reduced to a minimum, which is interesting for the transmission
- The aspect remains the same whatever the scale.

3.1.2 Edge detection

Most techniques for edge detection work using a convolution kernel in order to locally estimate the gradient magnitude. The gradient represents the intensity variation within the image. Following this convolution, the contours are recovered applying a threshold. These filters work well for general purpose, but are however very sensitive to noise. Moreover, resulting contours are usually regions of contours, i.e. there width is larger than one pixel, which is a problem for the stage of pixel chaining and polygonal approximation. The most popular edge detectors are Sobel [11], Roberts [13] and Prewitt [12]. Each use a slightly different kernel convolution from each other.

We used Canny edge detector [2] that tries to solve previously stated problem. It works in a multi-stage process. First of all, the image is smoothed by Gaussian convolution. Then we compute the gradient of the image to highlight high frequencies which often represent contours. The gradient is calculated by applying the two following convolution kernels.

$$Gx = \begin{bmatrix} -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} Gy = \begin{bmatrix} \frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix}$$

The gradient magnitude is then:

$$|G_{i,j}| = \sqrt{Gx(i,j)^2 + Gy(i,j)^2}$$

And its direction:

$$\theta_{i,j} = \arctan\left(\frac{Gy(i,j)}{Gx(i,j)}\right)$$

The next two stages are:

- *Non-maximal suppression*: We keep only the maximum in the gradient direction, this ensures to get one pixel wide contours.
- *Hysteresis*: An adaptive threshold using two criterions. It makes sure the same edge will not be broken by noise.

To summarize, involved parameters are: the size of the Gaussian kernel, its standard deviation, and two thresholds for non-maximal suppression. A result can be seen on figure 3.

3.1.3 Pixel chaining

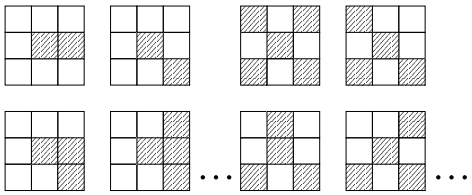
The pixels obtained from edge detection are then chained to obtain the poly-lines that will represent them. To this end, we create and fill a graph structure. Thus, we preserve the topology of edges, which seems necessary to a good comprehension of the picture.

First at all, we identify end-points and junctions of the pixel chains using 3×3 masks on the contour image. Basically, we run through the image and using masks like those depicted on figure 4, we are able to detect end-points and junctions. The detected pixels are the initial vertices of the graph. The resulting graph is weighted by the pixel chains between each vertex. These chains will be represented at the stage of vectorization by a poly-line. Starting from an end-point, we process the whole connex components following the contour in a breath-first traversal way. It still remains the components without any end-point: the loops, like the windows of the figure 5. A second traversal of the image is thus necessary. The starting pixel is this time the first unprocessed one. The figure 5 illustrates the different stages of this procedure.

The junctions identified using this method are not always positioned on the actual intersections between the segments. Indeed we seek those in the one neighbourhood, which does not allow to find their position with great accuracy. The figure 6 illustrates this problem. The figure 6(a) shows the detected junctions using the mask method. The real position of the intersection (figure 6(b)) can only be obtained knowing a vector representation of the segments involved. We obtain it easily after the polygonal approximation in the post-processing stage in merging locally close junctions.

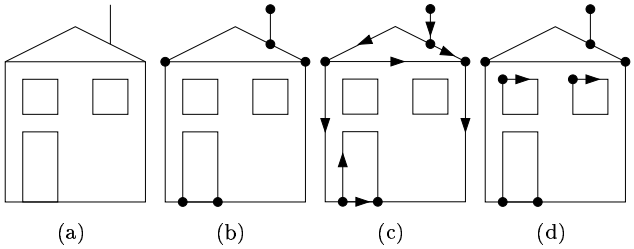
(a) Original image (b) Canny edge detector

Figure 3: Sample result applying canny edge detector



(a) End points (b) Junctions

Figure 4: Sample masks used to find initial vertices of the graph: (a) for end-point detection, (b) for junctions detection



(a) (b) (c) (d)

Figure 5: Stages for automatic stroke extraction: (a) Extracted edges. (b) End-points and junctions detection. (c) Breath-first traversal. (d) Loop processing.

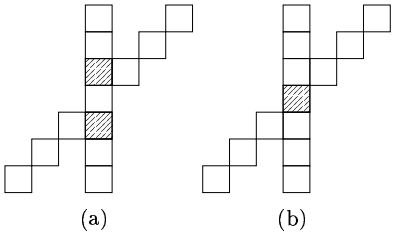


Figure 6: Problem when positioning junctions: (a) Detected junctions. (b) More likely position.

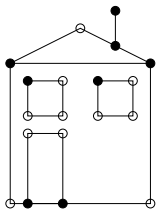


Figure 7: The graph after the polygonal approximation

3.1.4 Polygonal approximation

In this stage, each pixel chain of the graph is processed to obtain a poly-line representing it. The characteristic points obtained are then added as vertices in the graph structure. On figure 7 the vertices introduced by the polygonal approximation are white.

Polygonal approximation thus consists in transforming a chain of related points into a set of segments: a poly-line. Although several techniques exist on this subject, it seems that there is not any of them that makes reference. However we can classify them in two main families: merging methods and splitting methods.

The merging methods work in traversing the chain from the first point to the last. The points are merged in the same segment as long as the set of points checks a linearity criterion. The last point checking the criterion is the end-point of a segment and the start of the new one. This procedure functions in an iterative way.

The splitting methods works in a very similar way. Polygonal approximation is done this time recursively. The chain is initially approximated in a coarse way by only one segment connecting the two end-points of the chain. Then the segment is split in two in order to approach the chain as well as possible. The process is reiterated until all the segments check the criterion of linearity. The figure 8 shows an example of such an approximation.

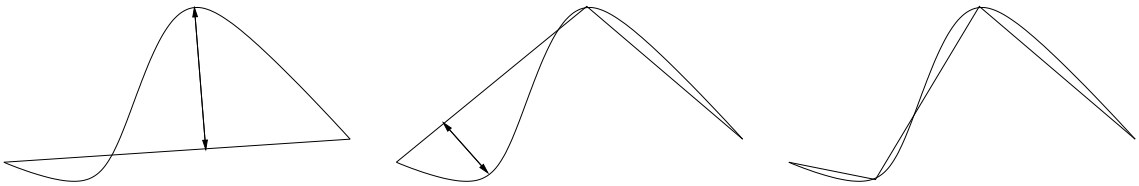


Figure 8: Splitting method using a distance-based criterion

Among the different algorithms we tested, the merging method using a distance-based criterion is the one that gave us the best results.

3.1.5 Post-processing and cleaning

In order to perfect vectorization, a post treatment is necessary. Its purpose is to simplify the approximation obtained and to correct the errors generated by the initial noise of the image and the stage of vectorization.

Post-processing steps we selected are:

- *Suppression of small size segments*: indeed, those are introduced by noise and thus do not bring anything to the legibility of the image.

- *Junctions merging*: as explained in the section 3.1.3, the method of the masks does not allow to find the right position of the intersections between the segments. The knowledge of a vectorial representation of the chains of pixels enables us to solve this problem in merging locally close junctions.
- *Straight lines detection*: in order to improve legibility, we seek to simplify the model as much as we can. Stating that the input image might have been distorted by perspective, the resulting edge might have too, and particularly the edge on straight walls or windows. In order to correct this, we detect aligned segments and merge them in one.
- *Vertical and horizontal line fitting*: As we work on photographs of buildings, we have a great number of horizontal and vertical lines. We seek here to redress these lines.

The number of lines removed at this step is evaluated in section 6.

3.2 3D modeling

3.2.1 Scene modeling

We now have a set of lines related to a facade. An easy way to get a 3D model from this data is to build a street in which each building is a simple bloc which size is obtained from the size of the associated image. However, the input images need to be at the same scale. This simplistic representation can be useful for testing purposes (see section 6).

We applied our method to a large model usable in a client-server configuration. We used the Bordeaux representation presented in [5]. This model generated from cadastral data is composed of a set of buildings and their related textures. Thus it is well suited for the deployment of the method we settled.

3.2.2 Line rendering

We can use the set of lines to represent a facade in different ways:

- The lines can be drawn in an image to be used in place of a texture. This image is indeed less complex than the original one and so offers a good compression rate using any image compression algorithm. We can consider using this as an easy way to try different stylization techniques
- A line can be represented as a geometric primitive on the surface of the building. It is possible to use a polygon to this end. However, these are not very usable for the representation of lines on a small screen. They cause many flickering trouble and result poor frame rates. Even so, we get encouraging results using 1 pixel wide lines.

Consequently, we tried mainly this latest method. However, we can consider different approaches depending on our objectives (ex: stylized rendering), or the evolutions of the devices. The figure 9 shows an example of rendering of a street using lines on Pocket PC. Rendering all lines without taking into account facade distance and orientation results a high density of lines in image space that declines legibility. This leads us to use level of detail techniques in order to solve this issue.

3.3 Level of detail

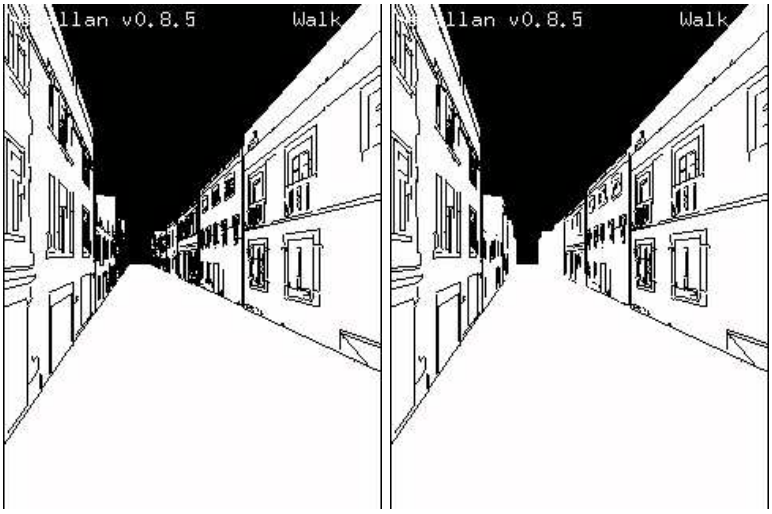
The lines previously found are rendered on the 3D model as lines of constant width whatever the distance they are viewed from. From a certain distance the screen is thus saturated with lines, decreasing the scene legibility. Level of detail technique gives a solution to this problem. Depending on the distance a building is viewed we show only a subset of the associated lines. The farther we are from a building, the smaller will be this subset. In this way we keep a reasonably line occupation density on the screen. Moreover, less geometric primitives are drawn improving the rendering time.

Given a set of lines associated to a building, an easy solution is to give a weight to each line to sort them in a hierarchical way. Thus we define the levels of detail as a set of distances:

$$D = (d_1, d_2, ..., d_n)\ \forall i,\ d_i \in [0, +\infty[\text{ and } d_i \leq d_{i+1}$$

related to a set of values:

$$V = (v_1, v_2, ..., v_{n-1})\ \forall i,\ v_i \in [0, 1[\text{ and } v_k \geq v_{k+1}$$



(a) Without level of detail (b) Using level of detail

Figure 9: A street rendered on a Pocket PC. (a) The display is quickly saturated with lines. (b) Using level of detail permits to reduce the line density.

In the distance interval $[d_k, d_{k+1}[$, we render only the lines which check $f(t) \leq v_k$. On the interval $[d_n, +\infty[$, we render the lines which checks $f(t) \leq v_n$. One might want the first level of detail to contain the whole set of lines, thus we define: $d_1 = 0$ and $v_1 = 1$. In the same way, if we do not want any line on the last level, we can define $v_n < 0$.

We can now sort the line in any kind of way according to their importance. We can choose to keep the longest lines first. Thus we define the following importance function on each line l of the set L. The $rank()$ function return the rank of the line in an array sorted according to the size of the lines.

$$f(l) = \frac{rank_{size}(L)}{card(L) - 1}$$

This way the v_k value represents a percentage of lines with a priority to the longest lines.

This approach gives satisfying rendering performances (see section 6). The process is however local, which is not suitable and can cause problems sometimes. We do not consider the closeness between the lines which can lead to unwanted local agglomerations of lines. We think of using the work of Barla and al. [1] on the simplification of line drawing.

4 Subdivision and visibility

The objective of this preprocessing step is to produce a VRML97 database that is compatible with the visibility streaming solution presented in [8]. It consists of an extension of VRML97 format allowing to describe visibility relationships for progressive transmission of 3D scenes.

This preprocessing stage can be decomposed in two steps; the first one producing the navigation space by subdividing the city model according to its topology and the second one computing the OPVS (Object Potentially Visible Set) of each cell of the navigation space. In order to produce such a database and to optimize the preprocessing step we rely on three different representations of the city model :

1. A single file that contains the polygonal representation of the entire city model. This representation is used to compute the navigation space.
2. A set of files, each one containing the partial description of a building (only its polygonal faces). This representation is used to perform the visibility computation.
3. A set of files, each one containing the full description of a building (its polygonal faces together with its vectorial appearance). After visibility computation, these files are substituted to the ones of previous representation to produce the final database.

4.1 Space subdivision

In order to generate the navigation space, we exploit the space subdivision algorithm presented in [7]. This algorithm takes a polygonal scenery (usually architectural) as an input and subdivides it into a set of cells (each cell containing its associated geometry) for which it establishes adjacency relationships. Such a subdivision result is usually used to compute cell-to-cells visibility relationships. In this case each cell refers to a set of potentially visible cells (CPVS) and thus to the geometry it contains. In our case we need a set of empty cells (following the topology of the urban scenery) in order to compute cell-to-objects visibility relationships. That is to say that each cell refers to a set of potentially visible objects (OPVS) that are buildings of the city. We thus use the algorithm in a slightly modified way.

First, we make the assumption that the ground of the urban scenery is flat (which is the case for our test scenery). The algorithm is then used as follows :

1. We produce a single cell that surrounds the entire polygonal model described in database 1 (see figure 10(a)).

2. We then cut this cell at a six meter elevation and we just keep the resulting cell placed between the ground and this cutting plane (see figure 10(b)). During visualization the user will thus be constrained to stay (to walk-through) within this slice of the whole space.

3. The constrained binary space subdivision is then performed on this cell. At each new cell subdivision the algorithm uses the geometry contained by the cell to find out the best plane to be used for its cutting. The convex volumes of the produced cells thus follows the topology of the scene (see figure 10(c)) which provides, at the visibility step, smaller PVS than if we used a regular subdivision such as an octree.

4. We finally delete all the geometry contained by the produced cells (see figure 10(d)).

The result of this process is a set of cells, following the topology of the city model, for which each cell contains a set of references to its adjacent cells.

4.2 Visibility preprocessing

The computation of the cell-to-objects visibility relationships is performed with the visibility algorithm presented in [9]. This algorithm is based on an OpenGL shooting system. Sample points are put in different places of the cell for which we want to compute the visibility relationships. We detect the set of cells and objects visible from theses points.

Basically, the scene is rendered six times for each sample point using a camera placed on the point and in the direction of each major axe. Such a group is called a render box, thus one render box is used for each sample point.

The detection of visible elements is performed in the following way. For each camera of each render box, the elements are rendered using their memory pointer as a color. Therefore, analysing the resulting image gives us the set of potentially visible objects.

This is not conservative but gives reasonable results using a small number of shooting boxes, usually one box per cell corner. More shooting boxes can be added if the result is not of sufficient quality (say some buildings misses). The advantage of this algorithm is that it is simple to implement and fast to execute since it uses the specialized graphics hardware.

As the visibility computation is performed by rendering the entire scenery several times we use the model described in database 2 (which is a simplified one). Indeed, appearance information is not useful to detect the visibility of a building. As an output of the algorithm the files containing the cells descriptions are completed with the visibility relationships.

4.3 Finalization

Files that contain the partial building description (the ones of database 2) are then substituted by files containing the full descriptions (the ones of database 3). As the URL are the same, the substitution is transparent and cell's references to objects are still valid. The entire set of files is then encoded using VRML97 binary format presented in [10] and compressed using the zlib.

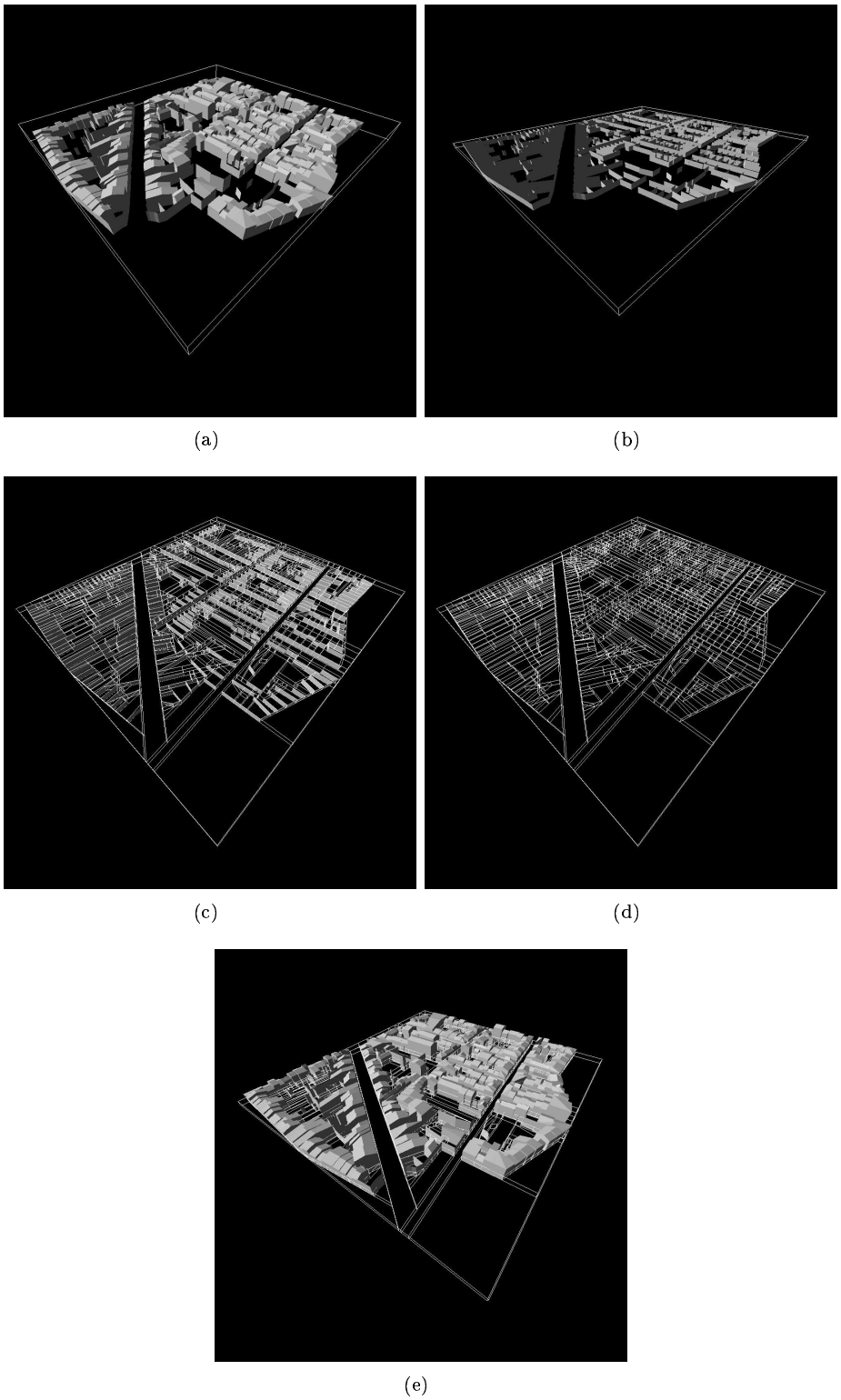


Figure 10: Subdivision steps. We start with a cell englobing the whole model (a). This cell is split up vertically and we keep the lowest 6 meter from the ground (b). We then apply a constraint BSP to get cells which follow the walls of the streets. The remaining geometry is removed (d) to be replaced by the original one (e).

5 Streaming and adaptive rendering

5.1 Visibility streaming

The process is done on-line on the client side. There are two stages which are processed in a sequential way: one loading step necessary for the navigation then a pre-fetching step to optimise the data transfers. These steps are depicted on figure 11.

The first step (figure 11(a)) occurs at the beginning of the visualization. The current cell is downloaded in a synchronous way together with its OPVS. Once this step finished, the user is able to begin the navigation in the cell and visualizes the elements of its OPVS. At this time, only the current cell and the potentially visible cells are available on the client. So, this one does not dispose of any information but the references on the adjacent cells to initiate potential pre-fetchings.

The second step (figure 11(b) and 11(c)) is divided in two.

Firstly it consists in downloading the adjacent cells of the current cell together with their OPVS in an asynchronous way and with a low priority. The adjacent cells are downloaded first because they contain the references on the elements of their own OPVS.

The second stage begins as soon as the first adjacent cell comes. This stage is based on a movement prediction of the user in order to detected a future visited cell. When such a cell is detected as future visited cell, all its adjacent cells together with their OPVS are downloaded in an asynchronous way. The algorithm caring of the movement prediction uses the last two positions of the camera without taking into account its orientation. Indeed, the camera orientation can evolve very quickly therefore do not reflect the actual displacement direction. Practically, we emit a ray from the oldest position to the latest one. All neighbour cells of the one with which the ray intersects are then used as future visited cells and so pre-fetched together with their OPVS.

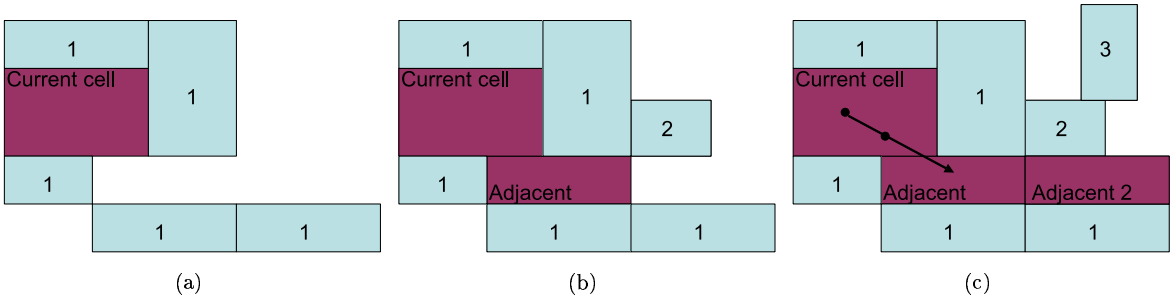


Figure 11: Loading steps: (a) Firstly the client fetches the current cell together with its OPVS. (b) Then it fetches the adjacent cell and its OPVS. (c) In the same time, the next visited cell is downloaded using a motion prediction algorithm.

5.2 Adaptative memory management

As the user is going through the scene, the client downloads progressively new cells. These cells are stored in memory so it will not need to download them again if the user comes back. The client builds up a graph using the adjacent relationships of the cells available on its side (figure 12). When the client needs a new cell but runs out of memory, it removes the cells the farthest to the current cell, i.e. the deepest in the graph, until enough memory is freed.

6 Results

We presented an alternative to textured rendering applied to urban scenery. We thus need to confront both methods: texture and line based rendering. Validation of the scene legibility depends on psychological matters

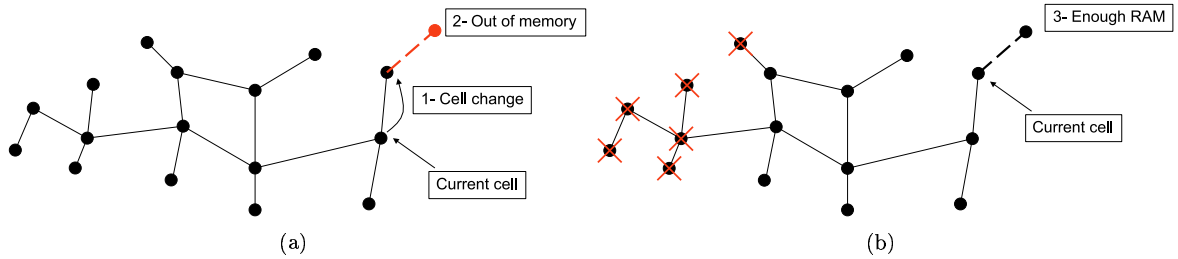


Figure 12: Memory management: When the client run out of memory and needs to fetches a new cell (a), it suppresses the deepest cells in the graph (b). The freed memory allows to fetch the new cell.

and is not the purpose of the current report. We compared line and texture based rendering on the following points: data size and rendering performances.

The rendering platform is a Toshiba Pocket PC (e800 serie) running Windows Mobile 2003. It is set up with a 400 Mhz ARM processor and 128 Mb of RAM memory. We use the Magellan client on Pocket PC to render VRML97 files. The Magellan client is built on the software OpenGL|ES library developed by Hybrid Graphics[2].

For the tests to be relevant, the data sets we are working on needs to be coherent. That is to say the images of facades must come from the same source (same model, same street, same acquisition device ...). Indeed we get comparable results applying the same parameter set on these kind of images. Obviously, it will not look homogeneous when mixing high and low resolution images or images from different cities.

Then the problem of data acquisition arises. The french website of the "pages jaunes[3]" offers images of facades of several towns. It is thus possible to pick up easily images from the same street. However, they are of very small resolution (250x320) and poor quality: the imperfections of JPEG compression are often visible. In order to confront the results to data of better quality, we have taken photographs of facades using a digital camera. The resulting pictures have a resolution of 3008x2000. From these input data we generated three streets using:

1. 10 images of the Borda street in Bordeaux coming from `www.pagesjaunes.fr`
2. 22 images of the Viadieu street in Toulouse coming from `www.pagesjaunes.fr`
3. 13 photographs of the cours de la Liberation at Talence taken with our digital camera

For the client-server application, we used the city model described in [5]. We processed the original textures. Despite the poor quality of the textures, we could so validate the approach in a client-server framework.

6.1 The post-processing step

This first test is the evaluation of the post-processing step described in section 3.1.5. It occurs after polygonal approximation and applies a set of heuristics in order to simplify the resulting lines. Figure 13 summarize the number of lines generated at the end of the polygonal approximation, and the post-processing step on each data set.

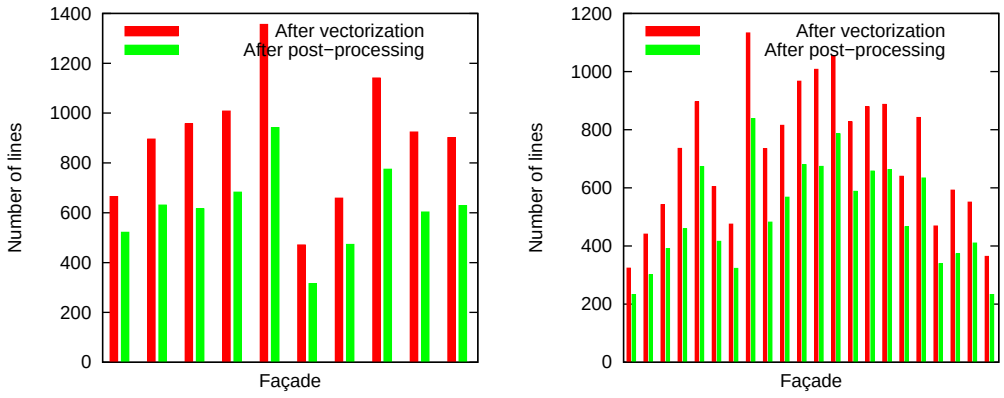
On the low resolution images (figure 13(a) and figure 13(b)), the post-processing step makes the number of lines decrease from 807 to 563 in average, so a gain of 30.15%. On the high resolution images (figure 13(c)), the number of lines decreases from 9628 to 4534 in average, with a gain reaching 54.96% this time. This step reduces dramatically the number of lines resulting from the polygonal approximation while keeping a similar look. Indeed the heuristics we chose are simple enough not to degrade too much the initial lines.

6.2 The street model

6.2.1 Data size

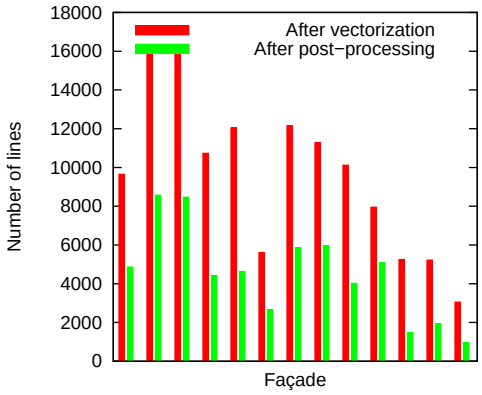
Comparison with the JPEG compression As an alternative to textured rendering, we have to compare the size of the generated lines with the size of a classic JPEG compression. The figure 14 presents the size of the

[2] `www.hybrid.fi`
[3] `www.pagesjaunes.fr`



(a) Borda street (10 images)

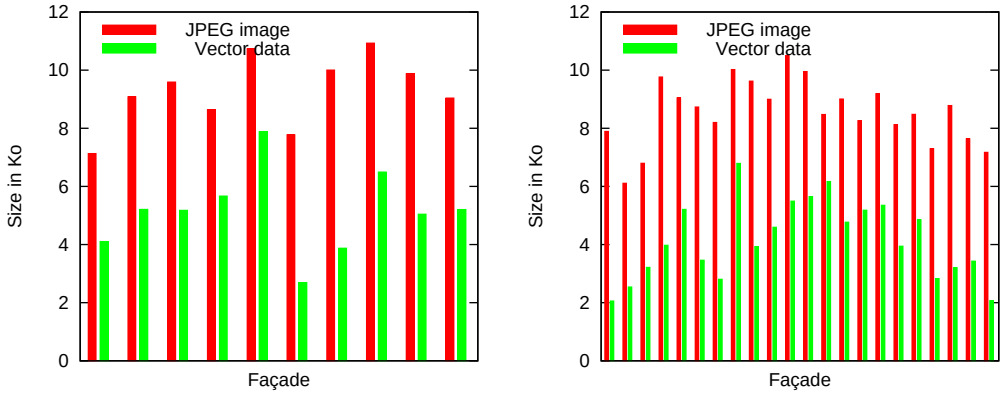
(b) Viadieu street (22 images)



(c) Cours de la Libération (13 images)

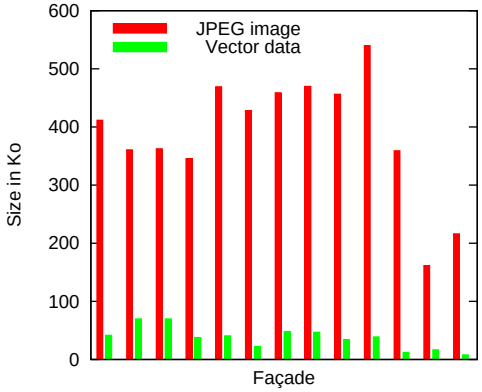
Figure 13: The number of lines is greatly reduced after the post-processing stage

vector files compared to the size of the JPEG files. All the sizes are in kilo bytes. The vector data are stored in a binary format and compressed using the zlib.



(a) Borda street (10 images)

(b) Viadieu street (22 images)



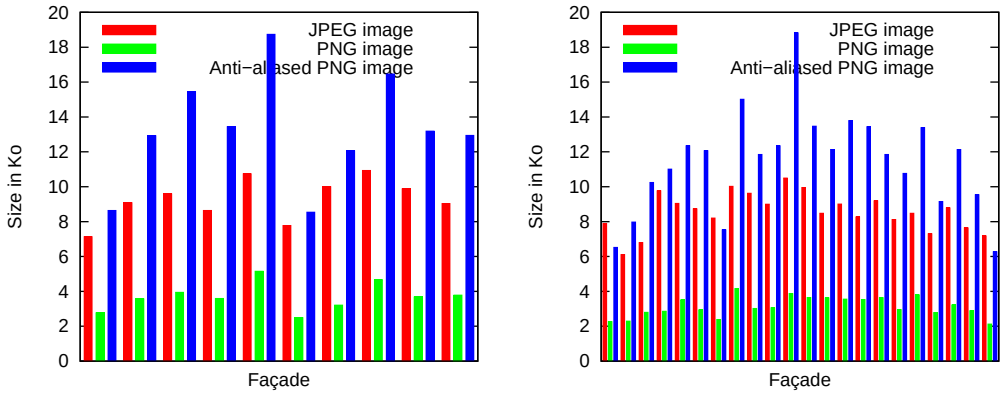
(c) Cours de la Libération (13 images)

Figure 14: Comparison between the size of vector data and JPEG images

The vector files resulting from the low resolution images (figure 14(a) and figure 14(b)) are in average two times smaller than the JPEG ones: 4.16 ko in average for the vector data compared to 8.55 ko for the JPEG compression. We have smaller data size using our method. Of course, the files contain far less information than compressed images. But we obtain a representation independent of the resolution. Oppositely an image depends strongly on its resolution. Thus for the high resolution images the ratio JPEG/vector is bigger than 10 (387.76 ko in average for JPEG compression compared to 37.55 ko for vector data).

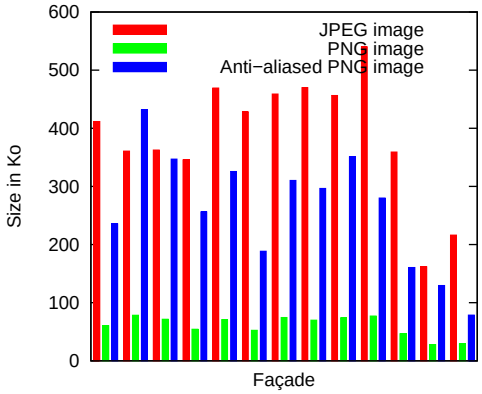
Comparison with drawn textures The lines can be drawn into an image to be used as a texture. Resulting images are less complex than original ones, so they offer a good compression ratio using any image compression algorithm. Such a representation is interesting because we can foresee using this approach to easily try several stylization methods. The figure 15 presents a comparison between the size of files compressed using JPEG and the size of the images in which we drawn the vector data. PNG format has been adopted instead of JPEG for this purpose because the JPEG method does not handle very well the compression of images that contain big color patches as our. Indeed, produced images have a white background on which the lines are drawn in black.

The resulting data sizes for PNG without anti-aliasing are fairly low, 3.13 kb in average for the low resolution images (figure 15(a) and figure 15(b)) and 60.56 kb for the high resolution ones (figure 15(c)). We can thus conceive to use them as textures. When an anti-aliasing is applied to the lines drawn, the color map grows as well as the size of the images. In high resolutions, resulting images are still smaller than the JPEG ones, 260.85 kb for PNG compared to 387.76 kb for JPEG in average. This is not the case for the low resolution images because of their poor quality.



(a) Borda street (10 images)

(b) Viadieu street (22 images)



(c) Cours de la Libération (13 images)

Figure 15: Comparison between JPEG images and vector data drawn into PNG images, optionally using anti-aliased lines

6.2.2 Rendering speed

The rendering tests presented in figure 16 has been done on the three data sets using textures, lines without level of detail and lines with level of detail on a the Pocket PC.

	Borda street	Viadieu street	Cours de la Libération
Number of lines	11188	6190	58949
Number of lines per facade	508.55	619	4534.54
Textured rendering	9.9 fps	8.2 fps	-
Rendering using lines	4.7 fps	3.2 fps	0.8 fps
Rendering using lines and level of detail	10.1 fps	9.6 fps	2.5 fps

Figure 16: Rendering performances on Pocket PC

Textured rendering is about two times faster than line based rendering. This is not surprising seeing the big number of geometric primitives used to render the scene. The frame rate increases using level of details to finally beats textured rendering. For the high resolution images, their use as textures is not practicable on Pocket PC, whereas using lines with level of detail the result is still interactive.

6.3 The city model

6.3.1 Data size

In order to test the different methods, we recorded a path in the scene and played it using the different methods: textures, lines without LOD, lines with LOD, and the geometry only. During the navigation, we recorded different statistics. The figure 17 shows the size of downloaded files using either the textures or the lines with LOD. The peaks visible the first minute are the files representing the facades: the lines and the textures. The remaining time only the geometry is transmitted and its size is fairly low and constant. If we had had more textures, the repartition of the peaks would be wider. As for previous tests using the street model, the lines are much smaller than the textures (7.55 ko in average for the lines, 35.61 ko for the textures).

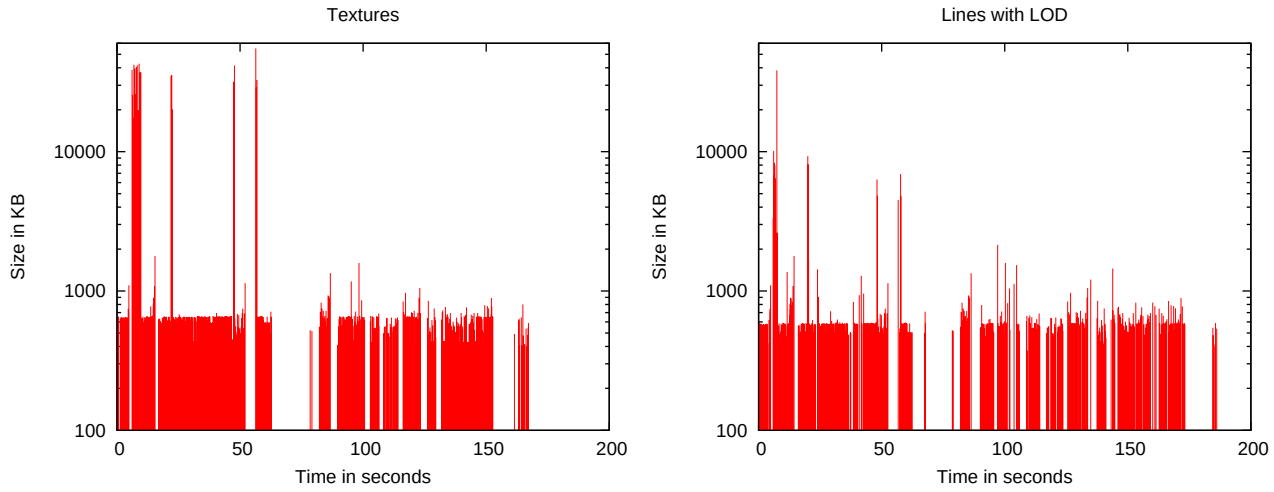


Figure 17: Comparison of the size of downloaded files during the streaming between the texture based model and the line based model

On figure 18, the plots represent the memory used on the client side using the different techniques. The geometry plot is the lower bound we can reach. As for the previous test, the memory used using the lines are much smaller than using the textures, and about the same as the geometry.

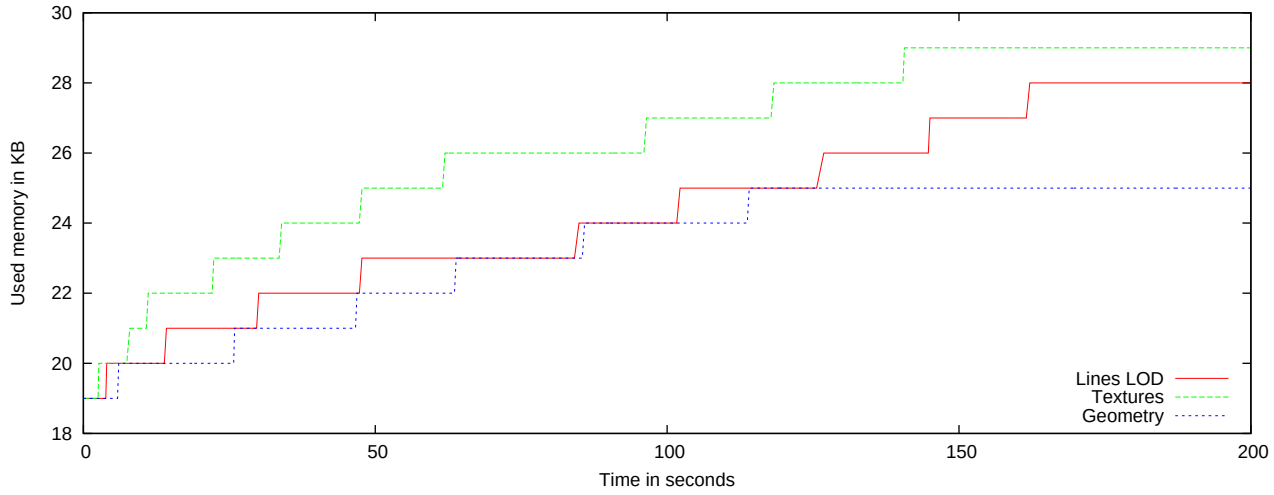


Figure 18: Comparison of the memory used on the client side during the streaming. Texture files are bigger than the vector one, consequently memory is more requested. The geometry plot gives the lower possible bound it is possible to reach.

6.3.2 Rendering speed

Figure 19 presents the rendering speed using each of the techniques. Once again, the geometry shows the higher frame rate we are able to reach. Using the lines without LOD results a frame rate of 1.5 fps in average, which is too slow to stay interactive. The rendering using the textures leads to good results (3.43 fps) and is still higher than using the lines with LOD (2.7 fps). However, the quality of the original textures are very poor and we are using non-connected lines. Further tests are needed using line strips and better quality textures in order to compare the performance of the different methods.

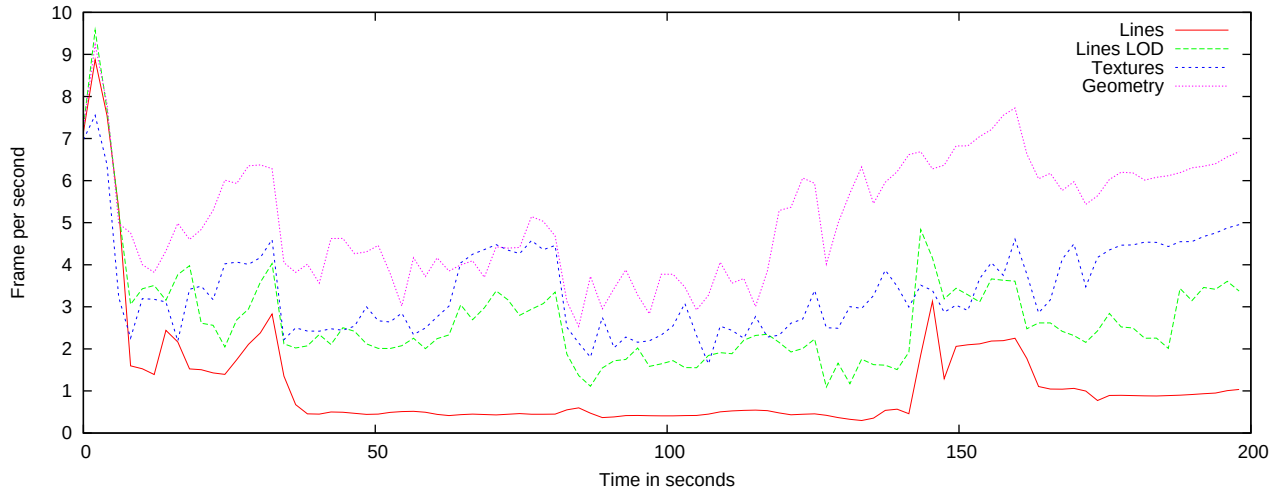


Figure 19: Framerate evolution during the navigation between the different tested methods.

The rendering steps is divided in two stages, the scene graph traversal and the rendering time. The next two figures (20 and 21) present the time spent in these two stages. The scene graph traversal time for the texture (22.21 ms) is lower than for the lines with LOD (102.93 ms). Indeed, the scene graph for the textures is simpler. Additionally, the node used for the lines contains a LOD node, and thus implies the computation of a square root at each frame.

However their rendering time is about the same: 215.51 ms for the textures, 246.39 ms for the lines with LOD. By implementing an optimised dedicated Magellan node, we would decrease the time spent in the graph traversal and consequently we would increase significantly the frame rate.

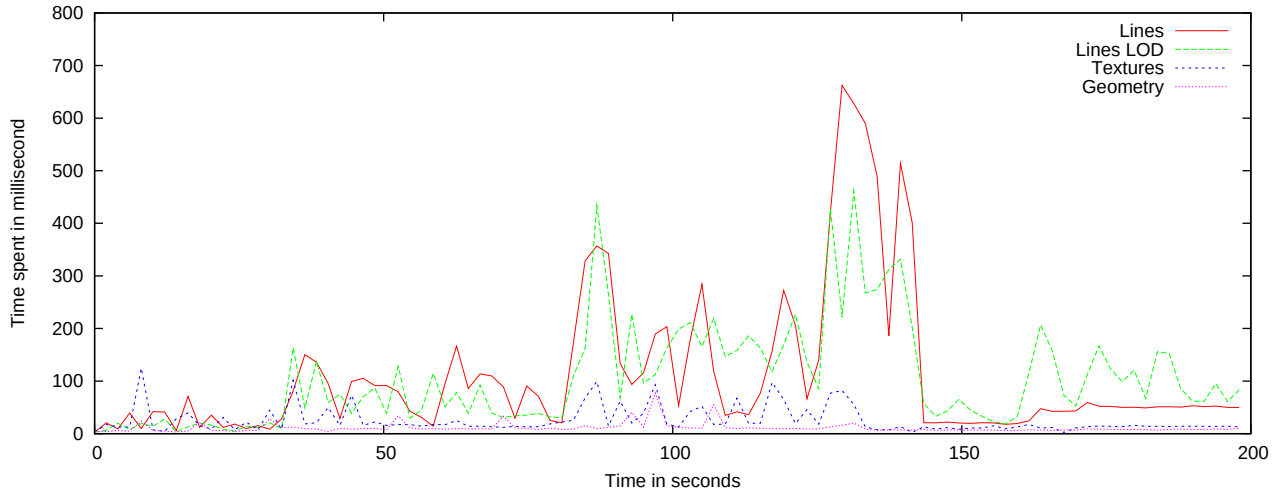


Figure 20: Time spent in traversing the graph scene using the line-based model, the texture-based one and the geometry only

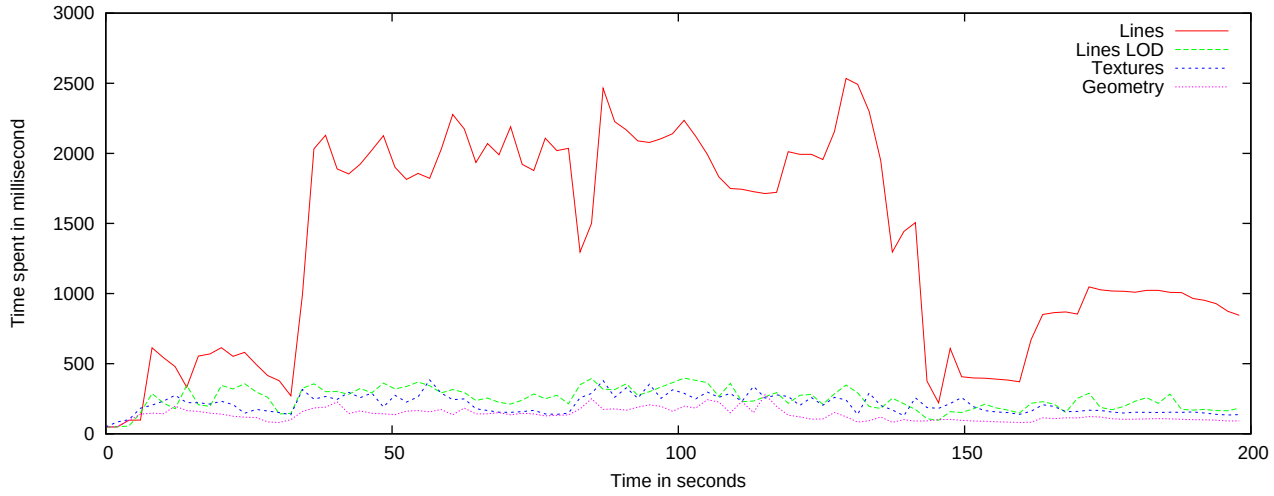


Figure 21: Time spent in rendering. Line-based rendering without LOD provoq greatly more time spent during rendering than other methods

6.3.3 Network

About the networking aspect, we measured the bandwidth on the figure 22 using the different methods. The bandwidth represents the rate the data are coming from the server to the moment they are available in the scene graph of the client, ready to be rendered. It includes: data transmission over the network, data decompression, VRML97 parsing, and node initialisation. Additionally, the Pocket PC uses 16 bits integers while those transmitted by the server are coded on 32 bits. So there is a 32 to 16 bits conversion on the fly on the client side. The texture node requires less time to be spent in parsing VRML and in converting integer and so it makes its bandwidth higher than for the lines. Even if there is less data to be transmitted, the time spent in all these processes make the bandwidth smaller for the lines. Once again, making a dedicated Magellan node would improve the result a lot.

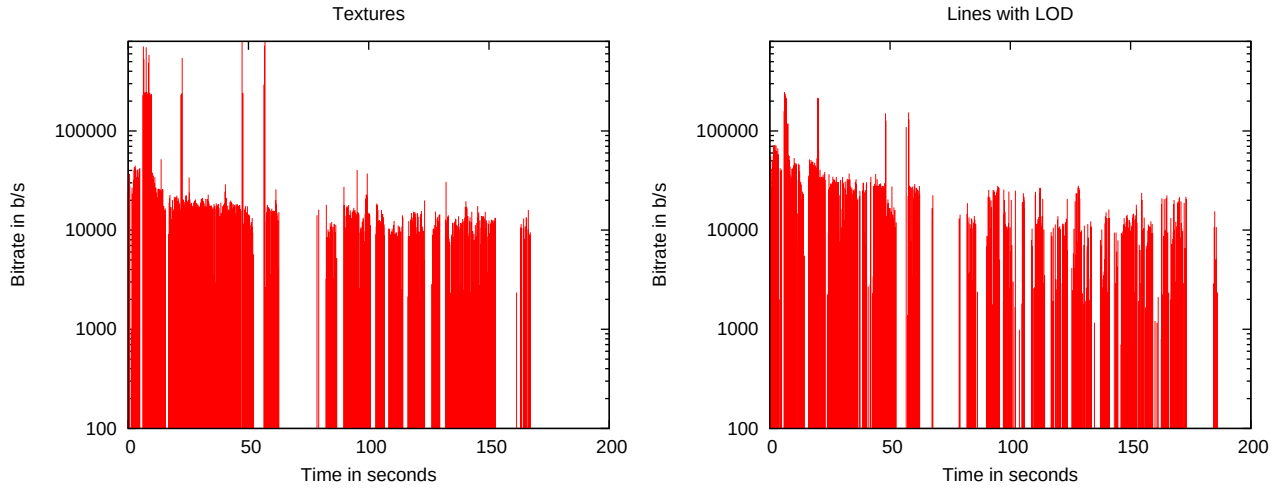


Figure 22: Bandwidth mesurement during the navigation using the texture based model and the line based one

7 Conclusion

We conceived non-photorealistic modelling and rendering techniques based on line for quick and expressive rendering of urban scenery. We implemented a rendering technique for facades based on feature lines extracted from photographs. The extraction is based on edge detectors; then the lines are resulting from polygonal approximation and post-processing. The line set is smaller than the original picture. This compact representation allows efficient data transmissions in a client-server application.

Using high quality textures is not feasible on Pocket PC. The feature lines extracted from these same images are however usable to represent a given facade. Our method thus offers a good compromise between legibility and efficiency.

This approach lets us foresee interesting prospects about transmission and rendering. The simplified representation we obtained could be used to test different stylization methods. Additionally the framework we settled could be exploited to render heterogeneous objects so mix up different rendering techniques in the same scene.

References

[1] Pascal Barla, Joëlle Thollot, and François Sillion. Geometric clustering for line drawing simplification. In *Proceedings of the Eurographics Symposium on Rendering*, 2005.

[2] J Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, 1986.

[3] Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, and Anthony Santella. Suggestive contours for conveying shape. *ACM Transactions on Graphics*, 22(3):848–855, July 2003.

[4] Doug DeCarlo and Anthony Santella. Stylization and abstraction of photographs. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 769–776, New York, NY, USA, 2002. ACM Press.

[5] Martin Hachet and Pascal Guitton. From cadastres to urban environments for 3d geomarketing. In *Proceedings of IEEE/ISPRS joint workshop on remote sensing and data fusion over urban areas*, pages 146–150, 2001.

[6] Aaron Hertzmann. Introduction to 3d non-photorealistic rendering: Silhouettes and outlines. In *SIGGRAPH 99*, chapter Course Notes. ACM Press, 1999.

[7] Jean-Eudes Marvie. *Remote Interactive Visualization of Complex Virtual Environments using Networks and Machines of Variable Performances.* PhD thesis, INSA de Rennes, 2004.

[8] Jean-Eudes Marvie and Kadi Bouatouch. A VRML97-X3D extension for massive scenery management in virtual worlds. In *Proceedings of the ninth international conference on 3D Web technology*, pages 145–153. ACM SIGGRAPH, 2004.

[9] Jean Eudes Marvie, Julien Perret, and Kadi Bouatouch. Remote interactive walkthrough of city models. In *proceedings of Pacific Graphics*, volume 2, pages 389–393. IEEE Computer Society, October 2003. Short Paper.

[10] Jean-Eudes Marvie, Julien Perret, and Kadi Bouatouch. Remote interactive walkthrough of city models using procedural geometry. Technical Report RR-4885, INRIA, July 2003. http://www.inria.fr/rrrt/rr-4885.html.

[11] K. K. Pingle. Visual perception by a computer. In *AII*, pages 277–284, 1969.

[12] J.M.S. Prewitt. Object enhancement and extraction. In B.S. Lipkin and A. Rosenfeld, editors, *Picture Processing and Psychopictorics*, pages 75–149. Academic Press, New York, 1970.

[13] L.G. Roberts. Machine perception of three–dimensional solids. In J. Tippet, D. Berkowitz, L. Clapp, C. Koester, and A. Vanderburgh, editors, *Optical and Electro-optical Information Processing*, pages 159–197. MIT Press, Cambridge, Massachusetts, USA, 1965.

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803